1. Briefly describe the theory of operation of the benchmark algorithm. Make sure you add a short description of its three main algorithms:
    a. Linked List: First you have 25% of a list, which has data and index separated from next and info pointers, located in memory one node after the other, and 75% of it, located randomly, to emulate the addition and deletion of nodes during some time.
        i. Then find operations are performed and record as part of the output chain.
        ii. Then, you get the list structure ordered by its data values, calculate the CRC and insert it in the output chain
        iii. Then, you get the list structure ordered by its index values, calculate the CRC and insert it in the output chain
        iv. Finally, list gets order again by its data values and the process can be repeated many times.
        v.
    b. Matrix Multiply: 3 two-dimensional matrices will fill up the entire memory. First matrix will only have numbers with 75% of the upper section of each word in zero (binary talking), second matrix will be filled with any numbers up to it mid-scale (50% upper bits in zero). Matrix C is used for storage of results. Then the following is performed:
        i. C = A x Constant
        ii. C = A x B [column vector]
        iii. C = A x B
        iv.
    c. State Machine: This is about exercising switch and if statements. It is used a state machine to identify string input as numbers and make division according to format. The separation of each number is given by a comma. The operations for this benchmark will be:
        i. To count every state transition and the final state after going over all the possible machine states.
        ii. Inject errors and repeat previous step.
        iii. Restore the input to its original form.
        iv.
2. How does the CoreMark benchmark try to deal with compiler optimization to come up with a standardized result? Make sure you include the next concepts in your description:
    a. Compile time vs run time

    b. Volatile variables

    c. Input-dependent results by using time based, scanf and command line parameters.

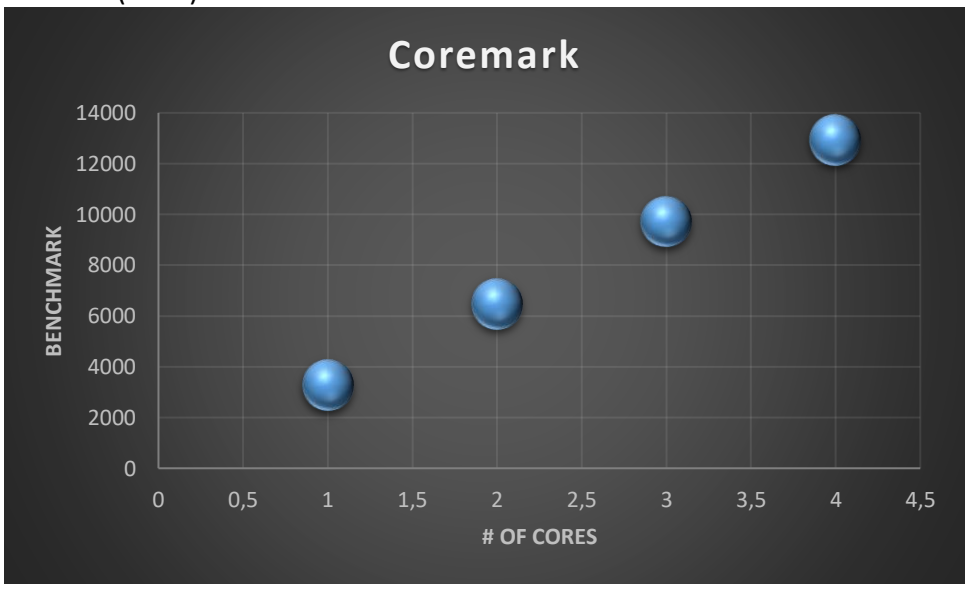
Using volatile variables help to code forcing the compiler to read it each time that it should be read. So, a way to standardize inputs to be read instead of precomputed by the optimizer of the compiler, is accomplished by mean of this type of variable declaration.
Another point to keep in mind is that command line parameters should not be used for a standardized benchmark since not all of the embedded systems have this capability.

3. What is the difference between the "core_portme" and the "core" files? Are we allowed to modify all of them?

4. Section 2.1.1.6: Set the ITERATIONS variable so that CoreMark runs at least 20 seconds in the RPI4. Use this value in all compilations of the next 2 sections.
   It was found to be Iterations = 64554 to get 20.003 seconds of duration.

5. Section 2.1.2: Experiment 1 -Multithreading
   a. Check how many threads your RPI4 can handle
      It was found to be 4 threads from here:
      Stackoverflow: Number of threads that Raspberry Pi can handle

   b. Plot your results in a "Benchmark Performance (y-axis) vs Number of Threads (xaxis)"



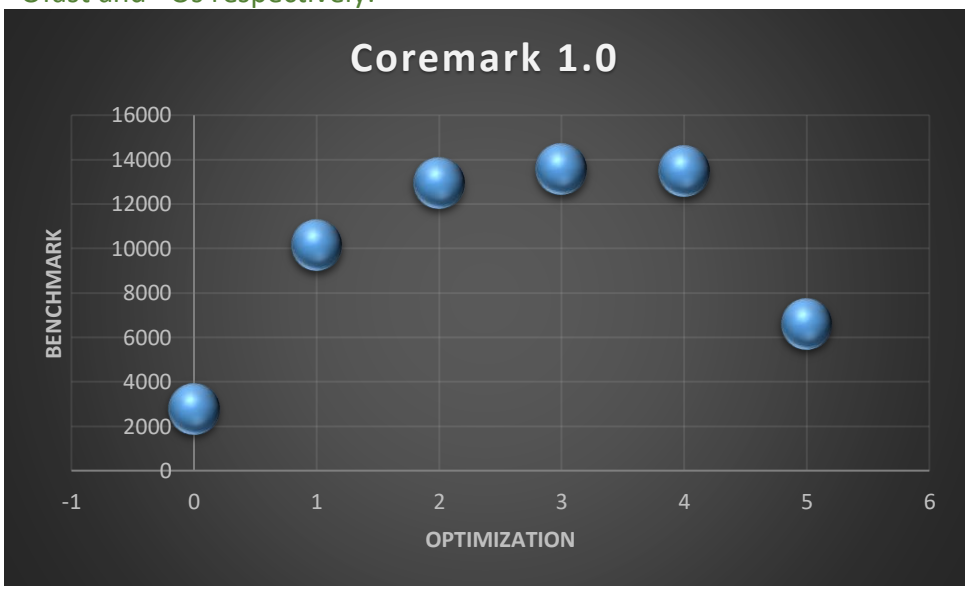
   c. Does your curve follow a linear, quadratic, exponential, etc behaviour or a combination of them? Which step produces the largest improvement in comparison to its predecessor?
      The curve follows a linear increase as the chart above is showing.

d. Would keep increasing the number of threads indefinitely be helpful to the benchmark performance?
After reaching the 4 threads, increasing number of these threads in the coremark has no effect on the final result. Result remains as the maximum possible amount of threads, above 12000 and below 13000.

e. What is the role the Linux scheduler plays in assigning threads to physical cores? Mention how the Linux scheduling policy works in your system.

6. Section 2.1.3: Experiment 2 –Compiler optimization
   a. Check the theory on compiler optimization and elaborate on what would be the best optimization according to the theory?
   In theory, -Ofast should give the best performance since it´s one step ahead of the fastest standard option during run time which is –O3. However, it´s also true that this –Ofast optimization only makes available a non-standard compliant programs that may or may not be on the current compiler.

   b. Compile the Benchmark with the following compiler optimizations and plot your results in a "Benchmark Performance (y-axis) vs Compiler optimization (x-axis)" graph.
   From 0 to 3 in x-axis means from –O0 to –O3 optimizations. 4 and 5 means –Ofast and –Os respectively:



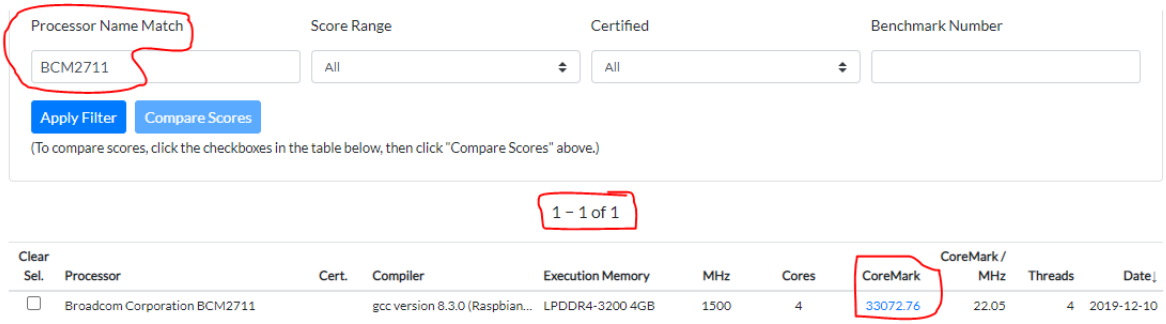
   c. Analyze your results, include at least the following points:
      i. Does your curve follow a linear, quadratic, exponential, etc behavior or a combination of them? Which step produces the largest improvement in comparison to its predecessor?
      In this experiment we clearly see a logarithmic curve when going from –O0 to –O3 optimizations. The biggest jump we get occurs

when going from –O0 to –O2, but is interesting to notice that –Ofast didn´t get any improvement which confirms that non-standard programs didn´t have any effect or they were just not implemented by the compiler.

ii. Is it possible to significantly improve the CoreMark benchmark with the compiler? Would you consider it a compiler-independent benchmark?
At this point of the experiment, becomes clear that CoreMark is directly and huge impacted by the optimization of the compiler. Therefore, we can conclude that this is NOT a compiler-independent benchmark at all.

d. Use the results of the last 2 sections to generate the best possible CoreMark result you can get out of the RPI4 with our custom Linux Kernel, File System and Toolchain.
The best possible CoreMark obtained from the last 2 sections is achieved when 4 threads are used, combined with optimization at –O3 level. Result obtained is 13562.48

e. Go to the scores list, find scores reported for the BCM2711 processor and compare yours. Analyze your results. Explain why it could be that your results are slower or faster than the reported ones.
Only one benchmark was found to be done during last month of 2019 in which the result give is as shown below:



As can be confirmed, 33072 in resultant benchmark has nothing to do with the 13562 obtained in our experiment. When going into software environment details of this benchmark, we found that compiler is different than the one we used:

Software Environment

Compiler Name and Version	gcc version 8.3.0 (Raspbian 8.3.0-6+rpi1)
Compiler Flags	-O2
Operating System Name and Version	Raspbian GNU/Linux 10 (buster)
Parallel Execution	4:Fork

The optimization level is even worst and the parallel execution was Fork with 4 threads. For the last difference, we tried again by swapping pthread with fork enabling on the core_portme.h file. Same result was obtained, around 13562, again. Compiler could be the root cause of the huge difference on the benchmark.

7. Section 2.2.1: Improved Algorithm
    a. How does the algorithm differ from the original one? What has improved?
    CoreMark-pro stresses not only the pipeline but the rest of the processor. It uses 5 integer and 4 floating-point workloads instead of just 1 and targets not only processor but also memory subsystems.

    b. Overview its integer and floating-point workloads without explaining in detail the 24 FORTRAN kernels.
    The 5 integer workloads are:
        1) JPEG compression
        2) ZIP compression
        3) XML parsing
        4) SHA-256Secure Hash Algorithm
        5) Memory-intensive extended from CoreMark

    The 4 floating-point workloads are:
        1) Radix-2 FFT.
        2) Gaussian elimination with partial pivoting derived from LINPACK.
        3) A simple neural-net
        4) Improved version of the Livermore loops benchmark using 24 FORTRAN kernels converted to C.

    c. Is the simple CoreMark included into the CoreMark-Pro?
    Yes, we can find in the datasheet.txt specifically stating as following:

Branch: master ▾ coremark-pro / benchmarks / core / datasheet.txt

petertorelli First import to GitHub

1 contributor

57 lines (48 sloc) | 2.64 KB

```
1   File: CORE
2   ==========
3           A benchmark derived from CoreMark with different datasets and a modified CRC algorithm.
4
5   Description:
6   ============
7   The original CoreMark was specifically built for portability
8   and to provide support for very small processors.
9   In the context of CoreMark-HPC, we have changed the CRC algorithm and the dataset,
10  to be a better fit for application processors.
```

d. How are the multiple workloads combined to summarize results in one single score?
   It is used a Perl script contained into this path (*util/perl/generate_summary.pl*) to compute each workload. This perl script summary is actually fed by cert_median.pl and results_parser.pl output results.

8. Section 2.2.2. Running the CoreMark-Pro
   a. The approach selected was to copy the whole source to RPI4 and compile it from there. We selected so because didn´t find a way to use the cross compile command ($CC…) successfully from the host. Instead, the command used directly on the target to compile and run the benchmark was:

      ⇨ make TARGET=linux64 SCMD='-c4' certify-all

   As specified on the readme of the source git code, it finally got what expected for a 4 context test with all the workloads executed:

```
WORKLOAD RESULTS TABLE

                                          MultiCore SingleCore
Workload Name                              (iter/s)   (iter/s)    Scaling
------------------------------------------ ---------- ---------- ----------
cjpeg-rose7-preset                            97.09      25.19       3.85
core                                           0.93       0.23       4.04
linear_alg-mid-100x100-sp                     93.46      24.19       3.86
loops-all-mid-10k-sp                           1.76       0.79       2.23
nnet_test                                      3.84       1.14       3.37
parser-125k                                    5.95       3.80       1.57
radix2-big-64k                                94.38      83.79       1.13
sha-test                                     156.25      47.17       3.31
zip-test                                      54.79      15.62       3.51

MARK RESULTS TABLE

Mark Name                                  MultiCore SingleCore    Scaling
------------------------------------------ ---------- ---------- ----------
CoreMark-PRO                                2146.72     778.11       2.76
```

b. Investigate if you can vary any compilation parameter as we did in Sections 2.1.3 and 2.1.2.

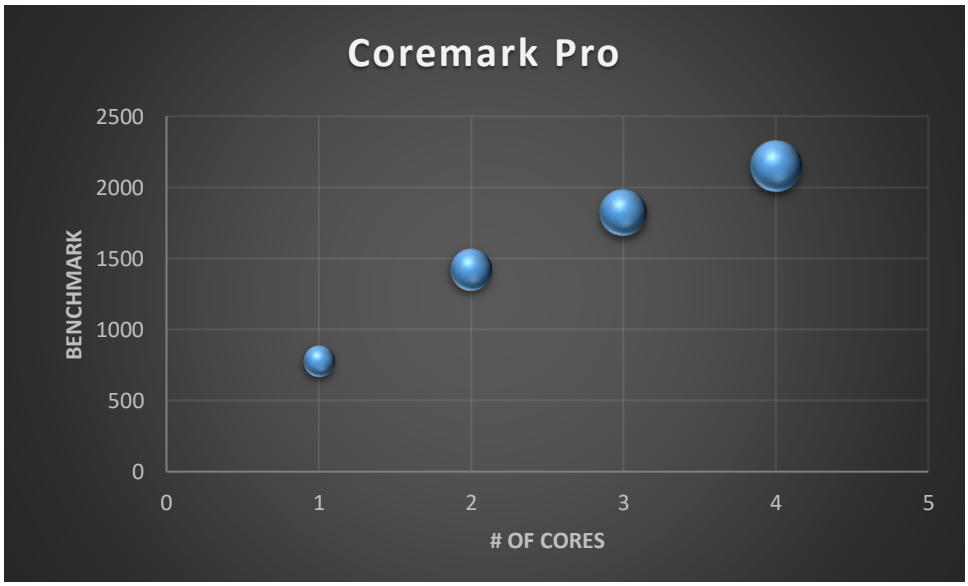
The only two parameters found to be passed to compilation are to define number of context and workloads to be executed. The result at the end is shown for multi-core and single core as can be confirmed from previous point “a”.

When running the full nine workloads for only 1 context, the result obtained is the following:

```
WORKLOAD RESULTS TABLE

                                                MultiCore SingleCore
Workload Name                                    (iter/s)   (iter/s)    Scaling
---------------------------------------------- ---------- ---------- ----------
cjpeg-rose7-preset                                  25.13      25.25       1.00
core                                                 0.23       0.23       1.00
linear_alg-mid-100x100-sp                           24.25      24.40       0.99
loops-all-mid-10k-sp                                 0.79       0.79       1.00
nnet_test                                            1.15       1.15       1.00
parser-125k                                          3.80       3.79       1.00
radix2-big-64k                                      83.30      83.06       1.00
sha-test                                            47.17      47.17       1.00
zip-test                                            15.62      15.62       1.00

MARK RESULTS TABLE

Mark Name                                       MultiCore SingleCore    Scaling
---------------------------------------------- ---------- ---------- ----------
CoreMark-PRO                                       778.37     778.84       1.00
```

…demonstrating in this way, something that seems very hard for the programmers to have placed in the readme file: that “***number of context***” means “***number of cores***”.

With that on mind, we are then able to plot results of this benchmark vs number of “contexts”:



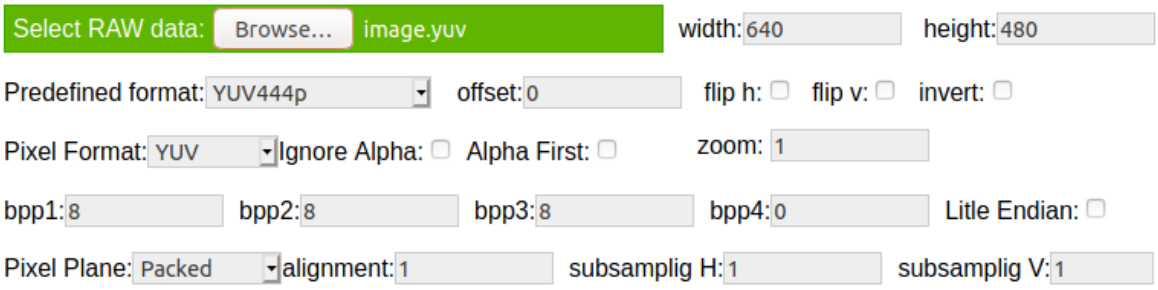
c. Based on your results and analysis answer the question: is the CoreMark-Pro a better benchmarking tool for our 64-bit, multi-core processor?

This is a better benchmark, since user doesn´t have the choice to modify the way this is to be compiled. Therefore, this is more compiler-independent that it predecessor CoreMark 1.0

9. Section 5: C implementation
    a. This will be the way to use new function:

```
Usage
./rgb2yuv [ -i RGBfile ] [ -o YUVfile ] [-h] [-a]
-i RGBfile specifies the RGB file to be converted.
-o YUVfile specifies the output file name.
-a displays the information of the author of the program.
-h displays the usage message to let the user know how to execute the
application.

Yocto prompt:
rgb2yuv -i image.rgb -o outputC.yuv
```

    b. These are the parameters that work well for the YUV file generated. This is for validation purposes:



Please pay special attention to the fact that this is a Packet Pixel Plane and not planar as suggested in the project manual.

    c. Benchmark and Analysis of implementation
        i. With original rgb2yuv function, time spent on conversion was around 24 +/- 3 [ms]. This is the implementation on code:

```
t=clock();
rgb2yuv(input_file, output_file);
t = clock()-t;
```

        ii. It was found here that **nice** values are user-space values that we can use to control the priority of a process. The nice value range is -20 to +19 where -20 is highest, 0 default and +19 is lowest. No change in elapsed time was found after using the following syntax in the command line, from any value from -20 to +19:
        @ nice –n [val] rgb2yuv –i image.rgb –o image.yuv

iii. According to this site, min/max values for the SCHED_FIFO priority are 1/99 respectively. FIFO priority was set to the maximum as requested by mean of:
@sudo chrt –f 99 ./rgb2yuv –i image.rgb –o image.yuv
But no change in the elapsed time was observed.

iv. It is asked to see for other possible settings but once consulted to Linux, these are the only chances:

```
SCHED_OTHER min/max priority    : 0/0
SCHED_FIFO min/max priority     : 1/99
SCHED_RR min/max priority       : 1/99
SCHED_BATCH min/max priority    : 0/0
SCHED_IDLE min/max priority     : 0/0
SCHED_DEADLINE min/max priority : 0/0
```

So, once we tried the RR option from 1 to 99 by the same procedure as before, no change on time was found. It is always executing conversion code around the 24 ms.

v. Analysis of results is summarized in the fact that no change in time was detected. Probably more information can be provided to avoid missing whatever is required to make these 2 commands (nice and chrt) to work well.

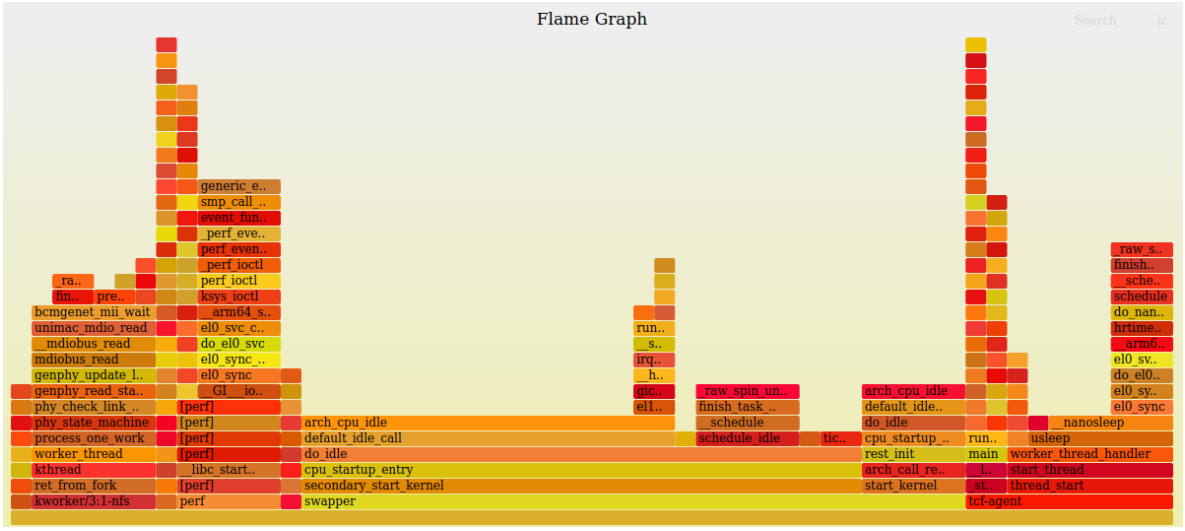
d. Turn off the Kernel Frequency Scaling.

i. There is just nothing underneath cpu0 folder. No frequency, no cpu_freq folder. Therefore, this point becomes obsolete in the guide document:

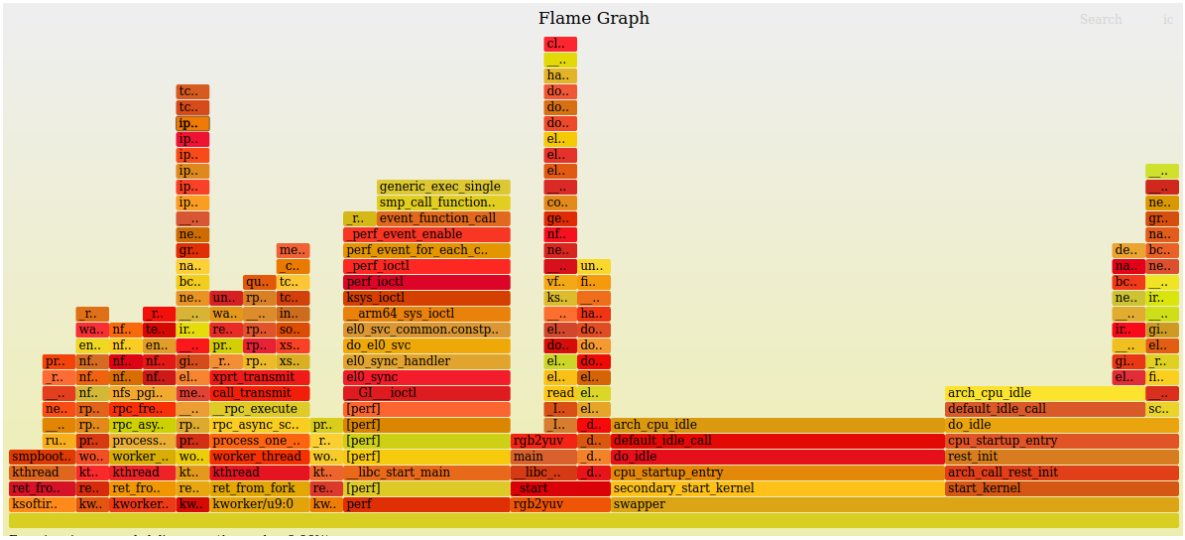
```
root@raspberrypi4-64:~# ls /sys/devices/system/cpu/cpu0/
cpu_capacity  of_node  power  regs  subsystem  topology  uevent
```

10.Profile your application

This is the result of sleep 60 profile



And this is the result with the rgb2yuv function



a. As it can be appreciated from rgb2yuv column, the biggest horizontal bar (called frame in Flame Graph) corresponds to the rgb2yuv function stacked over main stack call. It represents only the 5.71% of the total graph, but compared to the 8.57% of the rgb2yuv initial process, we can conclude that this is the main bottleneck of our application.
b. Other bottlenecks found are those related to IO streaming that we used to read and write the image input and output files respectively.
c. An obvious strategy to follow in this case will be parallelizing the conversion 'rgb2yuv' function as much as possible. We have implemented 2 'for' loops going through the whole input file that can be improved in this way. Another option to implement is to parallelize the output writing of the file along with the conversion of each pixel solving the second bottleneck found.

11. Section 7: C implementation with NEON intrinsics
    a. As this is a comparison between two different methods, we add to this report both core code examples:
        i. The regular code with no NEON intrinsic function:

```
void rgb2yuv(char* input_image, char* output_image){
        unsigned char RGB[3];
        FILE *inputFile, *outputFile;
        inputFile = fopen(input_image,"r");
        outputFile = fopen(output_image,"w");
        int input_count = fread(&file_buffer,sizeof(unsigned char), 921600, inputFile);

        for(int i=0; i<input_count; i+=3) {
                for(int j=0; j<3; j++) RGB[j] = file_buffer[i+j];
                file_buffer[i] = 0.257*RGB[0]+0.504*RGB[1]+0.098*RGB[2]+16;
                file_buffer[i+1] = -0.148*RGB[0]-0.291*RGB[1]+0.439*RGB[2]+128;
                file_buffer[i+2] = 0.439*RGB[0]-0.368*RGB[1]-0.071*RGB[2]+128;
        }
        int output_count = fwrite(file_buffer,sizeof(unsigned char), 921600, outputFile);
        fclose(outputFile);
        fclose(inputFile);
}
```

Time spent: 25 ms

```
root@raspberrypi4-64:~# rgb2yuv -i image.rgb -o image.yuv
Input RGB file image.rgb converted to output YUV file image.yuv in 0.029726 seco
nds
```

ii. The implementation code lend by professor using NEON:

```
// get RGB high and low vector sectors in 16bit uints (needed to handle multiplications)
uint8x8_t high_r = vget_high_u8(rgb_x3x16_tmp.val[2]);
uint8x8_t low_r = vget_low_u8(rgb_x3x16_tmp.val[2]);
uint8x8_t high_g = vget_high_u8(rgb_x3x16_tmp.val[1]);
uint8x8_t low_g = vget_low_u8(rgb_x3x16_tmp.val[1]);
uint8x8_t high_b = vget_high_u8(rgb_x3x16_tmp.val[0]);
uint8x8_t low_b = vget_low_u8(rgb_x3x16_tmp.val[0]);

int16x8_t signed_high_r = vreinterpretq_s16_u16(vaddl_u8(high_r, vdup_n_u8(0)));
int16x8_t signed_low_r = vreinterpretq_s16_u16(vaddl_u8(low_r, vdup_n_u8(0)));
int16x8_t signed_high_g = vreinterpretq_s16_u16(vaddl_u8(high_g, vdup_n_u8(0)));
int16x8_t signed_low_g = vreinterpretq_s16_u16(vaddl_u8(low_g, vdup_n_u8(0)));
int16x8_t signed_high_b = vreinterpretq_s16_u16(vaddl_u8(high_b, vdup_n_u8(0)));
int16x8_t signed_low_b = vreinterpretq_s16_u16(vaddl_u8(low_b, vdup_n_u8(0)));
```

Time spent: 56 ms

```
root@raspberrypi4-64:~# rgb2yuv_intrinsics -i image.rgb -o image.yuv
In file: image.rgb
Out file: image.yuv

-> Execution time: 0.056236
```

Unfortunately, because of the lack of time, it is no possible to go deeper in this case. It was supposed to be shorter in time but is not.

12. For any reason, the pthread library just don´t follow expected behavior, even after following the recommended link provided in the project manual:
    a. http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
    b. Code created based on this was:

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <math.h>

unsigned char file_buffer[921600];

void rgb2yuv(char* input_image, char* output_image);
void *rgb2yuv_Y(void* input_image);
void get_help();

int main(int argc, char **argv){
  pthread_t thread_Y, thread_U, thread_V;
  int iret_Y, iret_U, iret_V;
  clock_t t;
  char *input_file = NULL;
```

```
iret_Y = pthread_create(&thread_Y, NULL, rgb2yuv_Y, (void*)input_file);
t = clock()-t;

pthread_join(thread_Y, NULL);
printf("Input RGB file %s converted to output YUV file %s in %f seconds\

return 0;
}

void *rgb2yuv_Y(void* input_image){

}
```

The error just doesn´t allow us to keep going on:

```
/home/project2/Yocto/poky/meta-hpec/recipes-rgb2yuv/rgb2yuv/rgb2yuv_pthread-1.0/
src/rgb2yuv_pthread.c:105: undefined reference to `pthread_create'
/home/project2/Yocto/poky/meta-hpec/recipes-rgb2yuv/rgb2yuv/rgb2yuv_pthread-1.0/
src/rgb2yuv_pthread.c:108: undefined reference to `pthread_join'
```

This compiler simply ignores the pthread library. Our idea was to separate in 3 different functions the calculation of the YUV components since they don´t depend on each other to be computed.
Unfortunately, compiler doesn´t seem to be more cooperative to us on regards to pthread matter!

13. Finally, video is ready and uploaded as required.